Compiling cooperative multitasking of ABS to Scala

Andri Saar, Keiko Nakata
Institute of Cybernetics at Tallinn University of Technology
andri@cs.ioc.ee, keiko@cs.ioc.ee

Abstract

In order to prove the correctness of compilation for our new Scala-based distributed backend for ABS, we will focus on one aspect of the ABS language, namely cooperative multitasking inside concurrent object groups. We create a simplified language that includes this feature and then define a compilation function that targets a language based on Scala that supports delimited continuations and prove the correctness of the compilation function.

1 Introduction

The ABS language [2], developed by the HATS project, is an Abstract Behavioral Specification language for concurrent object-based systems, designed to straddle the gap between design-oriented and implementation-oriented specification languages. It allows high-level specification of a system, by abstracting over implementation details while remaining executable and providing flexible concurrency and synchronization mechanisms by means of asynchronous method calls, separation of objects into concurrent groups and cooperative scheduling for objects inside one group. As the language serves as a foundation for several different analysis, it also has a formally defined semantics.

The concurrency model of ABS can be split in two: in one layer, we have local, synchronous and shared-memory communication between objects in one concurrent object group (COG) and on the second layer we have asynchronous message-based concurrency between different concurrent object groups reminiscent of Creol [3]. The behaviour of one COG is based on the cooperative multitasking of external method invocations and internal method activations, with concrete scheduling points where a different task may get scheduled. Between different COGs only asynchronous method calls may be used; different COGs have no shared object heap. The order of execution of asynchronous method calls is not specified. The result of an asynchronous call is a future; callers may decide at runtime when to synchronize with the reply from a call. Asynchronous calls may be seen as triggers that spawn new method activations (or tasks) within objects. Every object has a set of tasks that are to be executed (originating from method calls). Among these, at most one task of all the objects belonging to one COG is active; others are suspended and awaiting execution.

As ABS is designed to be an executable language, there are several backends available that are able to execute programs: the reference implementation is defined in Maude, and there exists a Java-based backend designed to help in developing and debugging ABS models. We are creating a new backend for ABS, using the Scala language, that would provide true distribution to ABS by allowing different concurrent object groups to be executed on different nodes in the network. As the concurrency models of ABS and Scala are quite different, we employ the experimental continuations plugin for Scala [4] to implement cooperative multitasking within one thread of execution.

In this article, we focus on the first step of providing a provably correct code generator for ABS. We look at what happens inside one concurrent object group and define a simplified language based on ABS that lacks objects but retains asynchronous calls and cooperative

scheduling between tasks. For this language we define a compilation function into a target language inspired by Scala that lacks concurrency but provides delimited continuations and closures. We then proceed to show that after compiling programs in the source language to a program in the target language preserves the operational behaviour.

2 Implementing the ABS concurrency model in Scala

As the concurrency model of ABS is notably different from the one provided by Scala, we need to provide a runtime support library for our compiler that would provide the necessary communication primitives to the generated code.

The asynchronous message-based communication between concurrent object groups is similar to actors. In practice, objects may pass three kinds of values between COGs: (a) object references, (b) future references and (c) immutable data structures from the functional sublanguage. Our implementation has to guarantee that all of these values are easily serializable so they could be passed around over the network, and make sure that the references are unique in the whole cluster and addressable from any node.

Inside one COG we have cooperative multitasking with the control being released only at certain *scheduling points*: in particular, `await` e statements. The statement `await` e suspends the execution of the current task until the guard expression e evaluates to true. This statement is a scheduling point: the scheduler may decide to switch tasks and proceed with some other task even if the guard immediately holds [1]. In order to implement this efficiently we use continuations [1, 5]. Every time we reach a scheduling point, we capture the rest of the task in a continuation, store it together with its guard in the task set and give control back to the scheduler of the COG. The scheduler will then go through its list of tasks, evaluate the guards and picks a new task whose guard evaluates to true. If there are no such tasks, then none of the tasks in the COG are executable at that moment. In this case, the COG releases the execution thread until some external event causes some task to be ready for execution.

3 Compilation of simplified ABS

We have designed two languages, both endowed with small-step operational semantics, that are simpler than both ABS and Scala, yet exhibit the behaviour we are interested in. The source language models one COG in ABS by having asynchronous calls, await statement and cooperative multitasking. The target language includes primitive statements to manage the task set and supports continuations as the method to switch between tasks, which are used to implement a scheduler.

3.1 Source language

The source language is designed to be as minimalistic as possible while still retaining the interesting properties: thus, we have removed objects, COGs and futures from the language, resulting in a simple imperative programming language with asynchronous procedure calls and a way to wait for some condition to become true.

The statements S of the language are defined by:

$$S ::= x := e \mid \texttt{skip} \mid S_1; S_2 \mid \texttt{await}\ e \mid \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ e\ \texttt{do}\ S \mid f(\overline{e})$$

[1] This is a simplification: in ABS, if the guard immediately holds, we are not allowed to switch tasks.

$$\frac{\mathrm{env}_F \vdash n, \sigma \triangleright Ts' \to n', \sigma' \triangleright Ts''}{\mathrm{env}_F \vdash n, \sigma \triangleright Ts \parallel Ts' \to n', \sigma' \triangleright Ts \parallel Ts''} \qquad \frac{[\![e]\!]_{\rho',\sigma} = \texttt{true}}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle\rho\rangle \parallel n'\langle e, S, \rho'\rangle \to n', \sigma \triangleright n\langle\rho\rangle \parallel n'\langle S, \rho'\rangle}$$

$$\frac{}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle\texttt{await } e, \rho\rangle \to n, \sigma \triangleright n\langle e, \texttt{skip}, \rho\rangle} \qquad \frac{\mathrm{env}_F\ f = (\overline{x}, S) \quad \overline{v} = [\![\overline{e}]\!]_{\rho,\sigma} \quad n' \text{ fresh}}{\mathrm{env}_F \vdash n, \sigma \triangleright \langle f(\overline{e}), \rho\rangle \to n, \sigma \triangleright n\langle\texttt{skip}, \rho\rangle \parallel n'\langle\texttt{true}, S, [\overline{x} \mapsto \overline{v}]\rangle}$$

$$\frac{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle S_1, \rho\rangle \to n\langle S_1', \rho'\rangle}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle S_1; S_2, \rho\rangle \to n\langle S_1'; S_2, \rho'\rangle} \qquad \frac{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle S_1, \rho\rangle \to n\langle e, S_1', \rho'\rangle}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle S_1; S_2, \rho\rangle \to n\langle e, S_1'; S_2, \rho'\rangle}$$

Figure 1: Structural operational semantics for the source language (excerpt)

Besides statements, we have pure expressions e which consist of usual arithmetic and boolean expressions. The notation $\overline{e}$ (resp. $\overline{x}$ and $\overline{v}$) denotes a possibly empty sequence of expressions (resp. variables and values).

A program P consists of a function environment env_F which maps function names to pairs of a formal argument list and a statement, and global variable declarations with their initial values. The entry point of the program will be the procedure named *main*.

The scheduler may only switch between tasks when evaluating an `await` e statement or when the currently running task terminates. Procedure calls $f(\overline{e})$ return immediately with no result and the original task will proceed. Communication between different tasks is achieved via global variables (which may be seen as analogues to object fields in ABS).

During runtime, the program is represented as a *configuration* c that consists of an active task identifier n, a global variable mapping σ and a set of tasks Ts. A task has an identifier and may be in one of the three forms: (a) a triple $\langle e, S, \rho\rangle$, representing a task that is awaiting to be scheduled, where e is the guard expression and ρ is the local variable mapping; (b) a pair $\langle S, \rho\rangle$, representing the currently active task and (c) a singleton $\langle\rho\rangle$, representing a terminated task.

$$\begin{array}{lll} \textit{Configuration} & c & ::= n, \sigma \triangleright Ts \\ \textit{Task sets} & Ts & ::= n\langle e, S, \rho\rangle \mid n\langle S, \rho\rangle \mid n\langle\rho\rangle \mid Ts \parallel Ts \end{array}$$

The order of tasks in the task set is irrelevant: the parallel operator $\parallel$ is commutative and associative. Transition rules in the semantics are in the form $\mathrm{env}_F \vdash c \to c'$. A subset of the rules is shown in Figure 1. We assume given an evaluation function $[\![e]\!]_{\rho,\sigma}$, which evaluates e with respect to the local and global variable mappings ρ and σ.

In the semantics, we are interested only in a subset of the whole task set and the first rule allows us to focus on the subset that we are interested in. If the currently active task is in the form $\langle S, \rho\rangle$, then we proceed to evaluate the statement S. Otherwise, it has terminated, i.e., it is in the form $\langle\rho\rangle$, or it has reached a scheduling point, i.e., in the form $\langle e, S, \rho\rangle$. Then we may pick another task from the task set whose guard evaluates to `true`. When a procedure f is invoked, we evaluate the argument expressions $\overline{e}$ and put a new task $n'\langle\texttt{true}, S, [\overline{x} \mapsto \overline{v}]\rangle$ to the task set, where S is the body of the procedure, $[\overline{x} \mapsto \overline{v}]$ maps formal argument names $\overline{x}$ to their values $\overline{v}$ and n' is a fresh identifier. The execution will proceed in the active task and the new task may only get scheduled once we reach a scheduling point.

3.2 Target language

The target language is modeled on Scala. Instead of containing explicit support for scheduling points, the target language supports delimited continuations via the `shift` and `reset` statements and the scheduler is explicitly specified when compiling programs from the source language to the target language. In order to not clutter the language with arrays, we also

$$\frac{\text{env}_F \vdash \langle S, \rho, Ts, \sigma\rangle \to \langle S', \rho', Ts', \sigma'\rangle}{\text{env}_F \vdash \langle E[S], \rho, Ts, \sigma\rangle \mapsto \langle E[S'], \rho', Ts', \sigma'\rangle}$$

$$\frac{E \text{ does not cross } \texttt{reset}}{\text{env}_F \vdash \langle \texttt{reset}\ \{E[\texttt{shift}\ k\ \{S\}]\}, \rho, Ts, \sigma\rangle \to \langle \texttt{reset}\ \{S\}, \rho[k \mapsto [\texttt{reset}\ \{E[\texttt{skip}]\}, \rho]], Ts, \sigma\rangle}$$

$$\frac{}{\text{env}_F \vdash \langle \texttt{reset}\ \{\texttt{skip}\}, \rho, Ts, \sigma\rangle \to \langle \texttt{skip}, \rho, Ts, \sigma\rangle} \qquad \frac{(e, [S, \rho']) \in Ts \quad [\![e]\!]_{\rho', \sigma, Ts} = \texttt{true}}{\text{env}_F \vdash \langle \texttt{getTask}(), \rho, Ts, \sigma\rangle \to \langle S, \rho', Ts \backslash (e, [S, \rho']), \sigma\rangle}$$

$$\frac{\text{env}_F\ f = (\overline{x}, S) \quad \overline{v} = [\![\overline{e}]\!]_{\rho, \sigma, Ts}}{\text{env}_F \vdash \langle \texttt{addFunc}\ f\ \overline{e}, \rho, Ts, \sigma\rangle \to \langle \texttt{skip}, \rho, Ts \cup \{(\texttt{true}, [S, [\overline{x} \mapsto \overline{v}]])\}, \sigma\rangle} \qquad \frac{}{\text{env}_F \vdash \langle \texttt{return}\ \rho', \rho, Ts, \sigma\rangle \to \langle \texttt{skip}, \rho', Ts, \sigma\rangle}$$

$$\frac{[\![x]\!]_{\rho, \sigma, Ts} = [S, \rho']}{\text{env}_F \vdash \langle \texttt{invoke}\ x, \rho, Ts, \sigma\rangle \to \langle S; \texttt{return}\ \rho, \rho', Ts, \sigma\rangle} \qquad \frac{[\![x]\!]_{\rho, \sigma, Ts} = [S, \rho']}{\text{env}_F \vdash \langle \texttt{addClosure}\ e\ x, \rho, T, \sigma\rangle \to \langle \texttt{skip}, \rho, T \cup (e, [S, \rho']), \sigma\rangle}$$

Figure 2: Reduction semantics for the target language (excerpt)

include four primitives that handle adding and picking tasks from a task set. During runtime, we make use of closures and the `return` statement, which may only appear during evaluation but may not be used in programs. Closures are represented as pairs $[S, \rho]$ of a statement and a variable mapping. The special `return` statement allows us to restore local variable mappings. The language also contains the usual arithmetic and boolean expressions, and simple loop and branching statements, just as our source language.

$$\begin{array}{lll} \textit{Expressions} & e & ::= \ldots \mid [S, \rho] \\ \textit{Statements} & S & ::= \ldots \mid \texttt{reset}\ \{S\} \mid \texttt{shift}\ x\ \{S\} \mid \texttt{invoke}\ x \mid \texttt{return}\ \rho \end{array}$$

We introduce three primitive statements and one primitive expression for operating on the task set. A task is represented as a pair of a guard (a delayed expression) and a closure. The statement `addFunc` $f\ \overline{e}$ will add to the task set a pair of a guard `true` and a closure representing a call to the procedure f with $\overline{e}$ immediately evaluated and used as the arguments to the procedure. The statement `addClosure` $e\ x$ will add a pair of a guard e and the closure stored at x to the task set. The expression `isTask` evaluates to `true` if there are any task in the task set whose guard holds. The statement `getTask`() invokes a closure from the task set whose guard evaluates to `true`. When there are several valid tasks, which of the tasks will get invoked is unspecified. In our Scala backend, these primitives are implemented using a global array and first-class functions.

We specify the operational semantics of the target language using evaluation contexts [6]:

$$\textit{Evaluation contexts} \quad E \quad ::= [] \mid E; S \mid \texttt{reset}\ \{E\}$$

The configurations in the target semantics are in the form $\langle S, \rho, Ts, \sigma\rangle$ where S is the statement, ρ is the local variable mapping of the current task, Ts is the task set, containing pairs of a guard and a closure, and σ is the global variable mapping. The function environment env_F is a mapping from function names to pairs of parameter lists and statements, similar to the source language. The reduction rules are in the form $\text{env}_F \vdash c \mapsto c'$, and a subset of the rules is shown in Figure 2.

Due to the evaluation context, informally, we will always reduce the leftmost statement, ignoring the enclosing `reset`s. The `shift` statement is used to capture the continuation up to the nearest enclosing `reset`.

3.3 Compilation function

The compilation function generates a program in the target language for every program in the source language. The interesting part of the definition is given by

$$\begin{aligned}[\![\texttt{fun } f(\overline{x}) \texttt{ is } S]\!] &= \texttt{fun } f(\overline{x}) \texttt{ is } \{[\![S]\!]; \texttt{invoke } \mathit{sched}\} \\ [\![\texttt{await } e]\!] &= \texttt{shift } k\ \{\texttt{addClosure } e\ k; \texttt{invoke } \mathit{sched}\} \\ [\![f(\overline{e})]\!] &= \texttt{addFunc } f\ \overline{e}\end{aligned}$$

Every compiled program contains a prelude, which declares a global variable *sched* that will contain the scheduler continuation and a statement that adds the *main* function to the task set. The control is then passed on to the scheduler:

```
var sched : Closure = [skip, []];
addFunc main ();
reset {while isTask do shift k {sched = k; getTask()}}
```

The scheduler checks if there is any task ready to be executed (`isTask`); if there is, the rest of the computation (which means invoking the `while` loop again) is stored in the global variable *sched*. When the task terminates, the closure is invoked and the control passes back to the scheduler, which will proceed by trying to find another suitable task for execution. Similar pattern is followed with the `await` e statement: we store the rest of the execution in a continuation, add the continuation with its guard to the task set and pass the control back to the scheduler.

The following theorem states that the compilation function is correct: an one-step reduction in the source language is simulated by possibly multiple-step reduction in the target language after the compilation. To state the theorem formally, we extend the compilation function to configurations and define a similarity relation $\preceq$, which disregards the number of adjoining `reset`s, between configurations in the target language.

Theorem 1 (Correctness of compilation)**.** *For all configurations c_S, c'_S in the source language and c_T in the target language such that $c_S \to c'_S$ and $[\![c_S]\!] \preceq c_T$, there exists a configuration c'_T in the target language such that $c_T \mapsto^+ c'_T$ and $[\![c'_S]\!] \preceq c'_T$.*

4 Conclusion

We have designed a compilation function that compiles a simplified ABS-like language to a language based on Scala and showed that the resulting programs preserve the operational behaviour of the original program. This work can be used as a basis for extending it with other features currently omitted (such as futures, full objects and different COGs), gradually moving closer to ABS and its semantics as the source. The target semantics will also move closer to the actual implementation of the compiler and the Scala runtime. We also plan to formalise the proof with a proof assistant, such as Coq or Agda.

Acknowledgements

This research was supported by the EU FP7 project 231620 Highly Adaptable and Trustworthy Software (HATS) and the European Regional Development Fund (ERDF) through the Estonian Centre of Excellence in Computer Science (EXCS).

References

[1] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, LFP '90, pages 151–160, New York, NY, USA, 1990. ACM.

[2] Einar Johnsen, Reiner Hähnle, Jan Schäfer, Rudi Sclatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. Lecture Notes in Computer Science. Springer, to appear.

[3] Einar Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6:39–58, 2007. 10.1007/s10270-006-0011-2.

[4] Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. *SIGPLAN Not.*, 44:317–328, August 2009.

[5] Mitchell Wand. Continuation-based multiprocessing. In *Proceedings of the 1980 ACM conference on LISP and functional programming*, LFP '80, pages 19–28, New York, NY, USA, 1980. ACM.

[6] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115:38–94, November 1994.